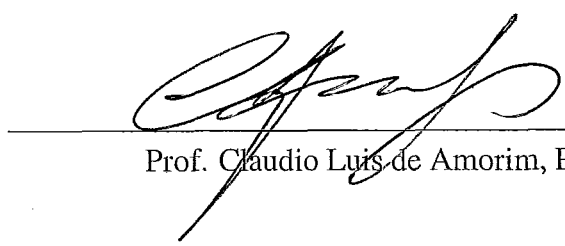
CLUSTERS JAVA: IMPLEMENTAÇÃO E AVALIAÇÃO

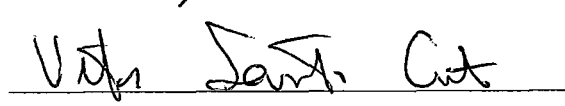
Anderson Faustino da Silva

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

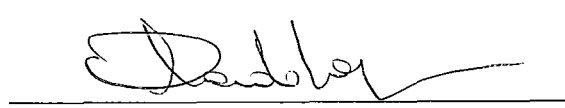
Aprovada por:



Prof. Claudio Luis de Amorim, Ph.D.



Prof. Vítor Santos Costa, Ph.D.



Prof. Orlando Gomes Loques Filho, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MAIO DE 2003

SILVA, ANDERSON FAUSTINO DA

Clusters Java: Implementação e Avaliação [Rio de Janeiro] 2003

IX, 56 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2003)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Computação de Alto Desempenho

2 - Java

3 - Memória Compartilhada Distribuída

I. COPPE/UFRJ II. Título (série)

Ao meu Deus, O Criador dos céus e da terra.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

CLUSTERS JAVA: IMPLEMENTAÇÃO E AVALIAÇÃO

Anderson Faustino da Silva

Maio/2003

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Nesta tese, propomos um novo ambiente de execução distribuída que denominamos cJava para executar transparentemente aplicações Java em *clusters*. Especificamente, cJava esconde a distribuição física das memórias dos nós do *cluster*, substituindo-a pela abstração de um única memória compartilhada distribuída. Para isso, cJava é capaz de distribuir as *threads* que compõem a aplicação Java pelas máquinas do *cluster* de maneira transparente ao programador. Nossa implementação de cJava requereu extensões significativas na máquina virtual Java original para fornecer a abstração proposta. Primeiro, um gerente de objetos distribuídos foi incorporado ao subsistema gerente de memória para criar um espaço global de objetos. Segundo, os acessos sincronizados ao espaço global de objetos foram estendidos para utilizar as primitivas *lock()* e *unlock()* fornecidas pelo sistema de memória compartilhada distribuída que cJava suporta. Terceiro, o subsistema de *threads* foi estendido para suportar criação remota e monitores globais. Finalmente, um subsistema responsável pela sinalização remota foi incorporado à máquina virtual Java. A principal propriedade de cJava é não requerer qualquer modificação da especificação da linguagem Java. Ainda mais importante é que os resultados experimentais de desempenho de cJava demonstram que a estratégia que esta tese defende para se construir clusters Java não somente é viável como também é altamente promissora.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

CLUSTERS JAVA: IMPLEMENTATION AND EVALUATION

Anderson Faustino da Silva

May/2003

Advisor: Claudio Luis de Amorim

Department: Computing and Systems Engineering

In this thesis, we propose a new distributed run-time environment, which we called cJava, for executing transparently multithread Java applications in *clusters*. Specifically, cJava is able to hide the physical memory distribution of the cluster's nodes by replacing it with the abstraction of a single distributed shared-memory. To this end, cJava distributes the threads of a Java application over the cluster's nodes in a transparent way to the programmer. Our implementation of cJava required significant extensions to the original Java virtual machine (JVM) to support the abstraction we proposed. First, a distributed object manager was incorporated to the memory management subsystem for creating the global object space. Second, synchronized accesses to the global object space were extended so that they could use the lock() and unlock() primitives the cJava's distributed shared memory system supports. Third, cJava extends the thread subsystem to support remote creation and global monitors. Last, a subsystem for remote signaling was added to the original JVM. The main property of cJava is that it does not require any modification to the Java Language specification. Most importantly, our experimental results of cJava's performance demonstrate that the strategy this thesis defends for building Java clusters is not only feasible but also highly promising.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Java [4] é uma linguagem de programação que tem recebido grande aceitação desde que foi criada e se tornou uma plataforma popular para o desenvolvimento de diversas classes de aplicações. Uma importante característica de Java é a portabilidade. O compilador utilizado pela linguagem Java gera código independente de arquitetura, permitindo que o mesmo código possa ser executado em diferentes arquiteturas de computadores, se houver a presença da máquina virtual Java para interpretar o código.

A linguagem também fornece um modelo *multithread* para programação concorrente, com grande potencial para aplicações distribuídas e paralelas. Entretando, a ausência de software especializado para processamento distribuído requer que o programador coordene a comunicação na camada da aplicação, usando algum mecanismo de comunicação entre processos baseado em passagem de mensagens (*sockets* [9], *rmi* [8], *corba* [6], etc).

Vários ambientes Java para computação de alto desempenho foram desenvolvidos [17], visando fornecer uma plataforma para computação distribuída e paralela capaz de retirar do programador a coordenação da comunicação. Java/DSM [30] foi a primeira proposta de um ambiente Java capaz de realizar computação de alto desempenho em plataformas distribuídas heterogêneas. MultiJav [5] é outro ambiente para plataformas heterogêneas, que possui ainda os atrativos de manter a portabilidade da linguagem Java e utilizar um compartilhamento baseado em objetos. cJVM [3] provê um ambiente com uma única imagem e fornece mecanismos para controlar o balanceamento de carga. Jackal

[26] implementa um ambiente de execução capaz de realizar busca antecipada de objetos.

Embora, o problema de coordenar a comunicação na camada da aplicação tenha sido retirado do programador, outros problemas surgiram na implementação desses ambientes: (1) em Java/DSM a localidade de uma *thread* não é transparente ao programador, (2) um substancial *overhead* é introduzido pelo ambiente MultiJav, (3) séria contenção na rede pode ocorrer em cJVM e (4) a semântica Java é alterada em Jackal.

Similarmente, cJava implementa a ilusão de um sistema multiprocessado como uma *interface* entre computação em *cluster* e o modelo de programação *multithread* Java. A principal idéia de cJava é encapsular os recursos distribuídos do *cluster* em uma camada de abstração, de tal maneira que a gerência dos recursos seja transparente à aplicação. Mais especificamente, nossa proposta é estabelecer esta ilusão modificando a máquina virtual Java, implementando uma máquina virtual distribuída que utiliza o modelo de memória compartilhada distribuída na comunicação entre processos e mantendo a semântica da linguagem inalterada. Esta estratégia foi escolhida para evitar qualquer modificação no sistema operacional ou na aplicação Java.

O ambiente cJava é implementado distribuindo-se as *threads* que compõem a aplicação pelas máquinas do *cluster* e suportando um repositório compartilhado de objetos. Com a ilusão de um sistema multiprocessado, as *threads* criadas pelo programador executam em paralelo acessando os objetos globais.

Uma questão importante da implementação de cJava foi tentar resolver os problemas apresentados pelos ambientes descritos anteriormente. Como ficará claro no decorrer do trabalho e principalmente nos capítulos 4 a 6, os problemas existentes nos outros ambientes não ocorrerão em cJava.

cJava possui as seguintes características:

- Encapsulamento do *hardware* utilizado. O *cluster* é encapsulado em um simples sistema de computação. Todas as *threads* criadas por uma aplicação podem ser

executadas em qualquer máquina do *cluster* e elas não precisam conhecer sua localidade física.

- Criação remota de *threads*. As *threads* criadas pela aplicação são automaticamente distribuídas pelo *cluster* para explorar o paralelismo.

- Compatibilidade. A implementação é compatível com o padrão da máquina virtual Java e as aplicações paralelas existentes podem ser executadas pelo sistema sem nenhuma modificação.

1.1 Contribuições da Tese

As principais contribuições desta tese são:

- A implementação e a avaliação do ambiente cJava para computação de alto desempenho utilizando Java em *clusters*.

- A apresentação de resultados experimentais que demonstram os pontos a serem otimizados para melhorar o desempenho do ambiente cJava.

- A demonstração que o ambiente cJava é capaz de fornecer um bom desempenho na execução de aplicações *multithread* e oferecer uma opção atrativa de ambiente de execução paralela Java.

1.2 Organização da Tese

O restante da tese está organizado da seguinte forma:

Capítulo 2 Descrição da estrutura interna da máquina virtual Java.

Capítulo 3 Apresentação dos conceitos básicos sobre memória compartilhada distribuída, e uma breve descrição sobre o sistema utilizado na análise de desempenho.

Capítulo 4 Descrição do ambiente cJava proposto.

Capítulo 5 Avaliação de desempenho de cJava.

Capítulo 6 Revisão dos principais trabalhos relacionados.

Capítulo 7 Apresentação de nossas conclusões.

chapterA Máquina Virtual Java

A máquina virtual Java é um computador abstrato [16], capaz de carregar classes e executar os *bytecodes*, que são instruções codificadas no formato da máquina virtual Java, nelas contidos. Ela é composta por três elementos: (1) um carregador de classe, que carrega as classes da *API* Java e as da aplicação a ser executada; (2) *heap*, uma região de dados que armazena as classes e (3) um motor de execução responsável por interpretar os *bytecodes* que implementam os métodos das classes. A figura 1.1 apresenta o diagrama de blocos da máquina virtual Java.

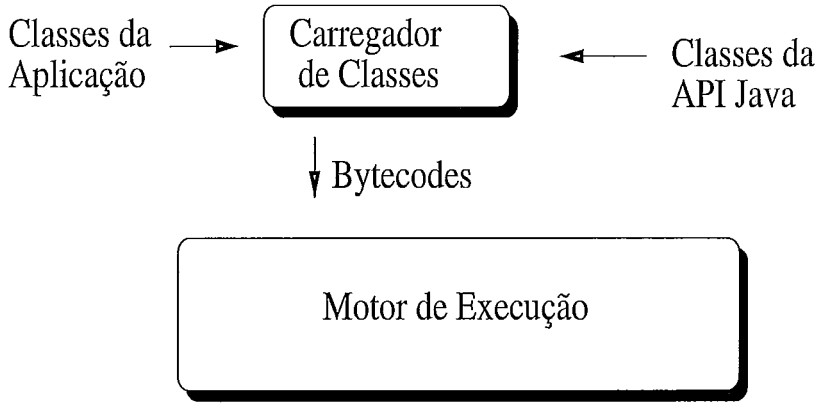


Figura 1.1: Diagrama de Blocos da Máquina Virtual Java.

A *heap* é uma área de dados, na qual todas as instâncias de classes e *arrays* são armazenados. A *heap* é criada durante a iniciação da máquina virtual Java. O armazenamento de dados na *heap* é gerenciado por um coletor de lixo, pois objetos Java não são desalocados explicitamente. A *heap* pode ter um tamanho fixo, expandir caso a aplicação crie vários objetos ou contrair quando objetos não são mais referenciados.

O motor de execução é a parte da máquina virtual que pode ter diferentes características. Um motor de execução simples apenas interpreta os *bytecodes*, um a cada unidade de tempo. Outro tipo mais veloz, porém requer mais memória, é o compilador *just-in-time*. Neste, os *bytecodes* de um método são compilados para código nativo durante a primeira invocação do método e armazenados em uma *cache* para um possível reuso, caso este método seja novamente invocado. Um terceiro tipo de motor de execução é um otimizador adaptativo. Para este, a máquina inicia interpretando *bytecodes*, porém a aplicação em execução é monitorada para detecção das áreas de código executadas com frequência (*hotspots*). Durante a execução da aplicação, a máquina virtual Java gera código nativo das áreas e continua interpretando os outros *bytecodes*.

Uma máquina virtual Java possui dois tipos de métodos: métodos Java e métodos nativos. Um método Java é escrito em linguagem Java, compilado para *bytecodes* e armazenado em arquivos de classes. Um método nativo é escrito em uma outra linguagem e compilado para código de máquina nativo de um particular processador. Métodos nativos são armazenados em bibliotecas dinâmicas. Quando um programa Java invoca um método nativo, a máquina virtual carrega a biblioteca dinâmica que contém a implementação do método e o invoca.

Este capítulo fornece uma breve descrição dos principais componentes da máquina virtual Java: o carregador de classes, a *heap* e o motor de execução.

1.3 O Carregador de Classes

A máquina virtual Java possui uma arquitetura flexível para carregadores de classes, permitindo uma aplicação Java carregar dinamicamente classes de diversas maneiras.

O carregador de classes [15] é responsável não apenas por localizar e importar os dados binários das classes. Ele dever ser capaz de verificar a corretude dos dados importados, alocar e iniciar memória para as variáveis das classes e resolver as referências

simbólicas. Estas atividades são realizadas na seguinte ordem:

1. Carregar: encontrar e importar os dados binários para um dado tipo.
2. Ligar: executar a verificação, a preparação, e opcionalmente a definição.
   (a) Verificação: assegurar a exatidão do tipo importado.
   (b) Preparação: alocar a memória para variáveis da classe e iniciar a memória com os valores padrões.
   (c) Definição: tranformar referências simbólicas em referências diretas.
3. Iniciar: invocar o código Java que inicia as variáveis da classe com seus valores apropriados.

Uma aplicação Java pode fazer uso de dois diferentes carregadores de classes: um carregador padrão do sistema, que é um componente da implementação da máquina Java e um carregador definido pelo usuário, capaz de carregar classes por meios não convencionais. Enquanto o carregador padrão é uma parte da implementação da máquina Java, o definido pelo usuário é uma classe Java carregada pela máquina virtual Java e criado como um outro objeto. Veja figura 1.2.

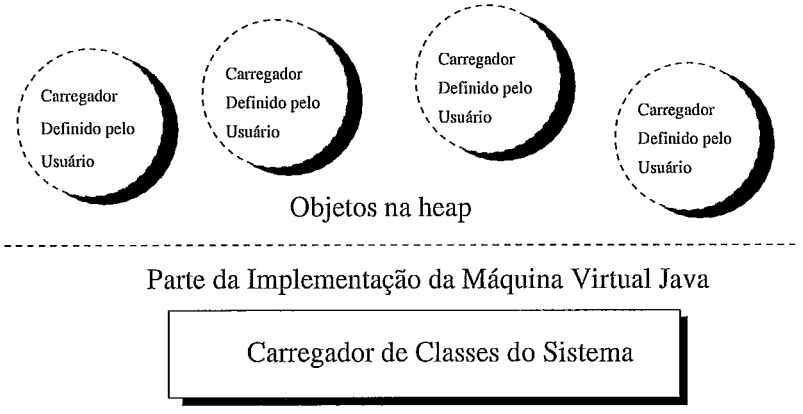


Figura 1.2: Arquitetura do Carregador de Classes.

Para cada classe carregada, a máquina virtual mantém um histórico contendo qual carregador carregou a classe. Quando uma classe faz referência a outra classe, a máquina

virtual Java utiliza para a carga da classe referenciada o mesmo carregador da classe que contém a referência. O uso deste mecanismo supõe que uma classe apenas referencie classes carregadas pelo mesmo carregador.

1.4 A Heap

Em uma instância da máquina virtual Java, informações sobre os tipos carregados são armazenadas em uma área lógica da memória denominada área de métodos [27]. Quando um tipo é carregado, a máquina utiliza um carregador de classe para localizar o arquivo de classe apropriado. A tarefa do carregador é ler o arquivo e passá-lo para a máquina virtual Java. Esta, por sua vez, extrai informações sobre o tipo e as armazena na área de métodos.

A área de métodos é compartilhada por todas as *threads* da aplicação. Porém, quando duas *threads* tentam encontrar uma classe que ainda não foi carregada, apenas uma carrega a classe, enquanto a outra fica bloqueada em espera.

Quando uma aplicação Java em execução cria uma instância de uma classe ou de um *array*, a memória para este novo objeto é alocada em uma *heap*. A máquina Java possui apenas uma *heap*, que é compartilhada por todas as *threads* da aplicação. Estas podem acessar os dados contidos na *heap* pertencente a outra *thread*. Para evitar condições de corrida, aplicações Java devem utilizar primitivas de sincronização.

A máquina virtual Java possui uma instrução que aloca memória na *heap* para um novo objeto, porém não possui uma instrução para liberação de memória. A liberação das áreas de memória ocupadas por objetos que não são referenciados pela aplicação é de responsabilidade da máquina virtual Java.

Um coletor de lixo é utilizado pela máquina virtual Java para gerenciar a *heap* e a área de métodos. O coletor de lixo, além de liberar as áreas utilizadas pelos objetos e classes que não estão sendo referenciados, possui a capacidade de realocar classes e suas

instâncias para reduzir a fragmentação da área de métodos e/ou da *heap*.

O tamanho da área de métodos e da *heap* não são fixos. Durante a execução de uma aplicação Java, a área de métodos ou a *heap* podem ser expandidas ou contraídas pela máquina virtual Java.

1.5 O Motor de Execução

O núcleo da máquina virtual Java é seu motor de execução [27], cujo comportamento é definido através de um conjunto de instruções.

Cada *thread* da aplicação é uma instância do motor de execução. Durante o período de vida da *thread*, ela está executando *bytecodes* ou métodos nativos. A *thread* pode executar *bytecodes* diretamente, interpretando, ou indiretamente, usando compilação *just-in-time* e executando o código nativo resultante.

A máquina virtual Java pode usar *threads* que não são visíveis à aplicação, por exemplo, a *thread* que executa a coleta de lixo. Tais *threads* não precisam ser instâncias do motor de execução. Entretanto, as *threads* que pertencem à aplicação são motores de execução em ação.

O Conjunto de Instruções

Uma seqüência de *bytecodes* é uma seqüência de instruções. Cada instrução consiste de um código de operação seguido por zero ou mais operandos. O código de operação indica a operação a ser executada. Os operandos fornecem os dados necessários para executar a operação especificada. No próprio código de operação existe uma indicação da existência ou não de operandos e dos mecanismos para acessá-los. Muitas instruções não fazem exame de nenhum operando e consistem somente em um código de operação.

O motor de execução funciona executando *bytecodes* de uma instrução por vez. Este

processo ocorre para cada *thread* da aplicação. O motor de execução busca um código de operação e se esse código possuir operandos, busca os operandos. Executa a ação solicitada pelo código e em seguida busca um outro código. A execução dos *bytecodes* continua até que uma *thread* termine retornando de seu método inicial.

Cada tipo de código do conjunto de instruções da máquina virtual Java possui um *mneumônio*, no estilo típico de linguagem *assembly*.

A máquina virtual Java não possui nenhum registrador para armazenar valores temporários. Cada método possui um conjunto de variáveis locais. O conjunto de instruções trata as variáveis locais como um conjunto de registradores que são consultados pelos índices.

Threads

Uma vantagem em usar *threads* em uma máquina multiprocessada é que as diferentes *threads* de uma aplicação podem ser executadas simultaneamente por diferentes processadores.

Uma *thread* Java pode ser executada com qualquer prioridade fornecida pela máquina virtual Java ou pelo programador. A máquina virtual Java possui um conjunto de dez prioridades, sendo a prioridade 1 (um) a mais baixa e a prioridade 10 (dez) a mais elevada. As *threads* com prioridade baixa começarão sua execução somente quando todas as *threads* de prioridade mais elevada estiverem bloqueadas. A máquina virtual Java usa prioridades para obter informações sobre quais *threads* devem possuir uma maior fatia de tempo do processador.

As *threads* da máquina virtual Java suportam dois tipos de sincronização: *lock* de objeto e *thread wait* e *notify*. *Lock* de objeto permite que não existam interferências entre as *threads* que acessam objetos compartilhados. *Wait* e *notify* permitem que as *threads*

cooperem entre si para alcançarem um objetivo comum.

Cada objeto alocado na *heap* é associado a um monitor que controla o *lock* do objeto. Condições de corrida são eliminadas através do uso de métodos sincronizados. Tais métodos devem obter acessos exclusivos à um monitor para acessar um determinado objeto.

Uma *thread* pode executar sobre um monitor duas operações distintas, que não são acessíveis ao programador: *monitor enter* e *monitor exit*. Se uma *thread* invoca um método sincronizado, porém não pode ganhar o acesso exclusivo ao monitor, ela é colocada na fila de espera do monitor. Isto ocorre quando outra *thread* está utilizando o monitor. O monitor também possui uma fila condicional para conter as *threads* que entraram no monitor porém invocaram um *wait* para aguardar uma mudança de estado em um campo. Uma ou mais *threads* são transferidas da fila condicional para a fila de espera quando o proprietário corrente do monitor invoca um *notify* ou *notifyall*, respectivamente. Quando a *thread* proprietária do monitor realiza um *monitor exit*, uma *thread* é removida da fila de espera e colocada na fila de prontas, fila que contêm as *threads* que estão aguardando seu momento de execução, para posterior execução.

Capítulo 2

Memória Compartilhada Distribuída

Um sistema de memória compartilhada distribuída [10] implementa o modelo de memória compartilhada em sistemas com memória fisicamente distribuída. O uso de memória compartilhada distribuída foi proposto para dar suporte a programação distribuída e paralela em redes de estações de trabalho, possibilitando que processos em diferentes máquinas compartilhem um espaço de endereçamento comum.

O modelo fornece uma *interface* transparente e um conveniente ambiente de programação. A vantagem deste modelo de programação é bem conhecido. Compartilhar memória torna transparente a comunicação entre processos e os programas escritos são usualmente mais fáceis de se entender. Além disto, com um espaço de endereçamento global, o programador pode focar no desenvolvimento do algoritmo ao invés do particionamento dos dados e da comunicação entre processos.

Sistemas de memória compartilhada distribuída utilizam mecanismos de *hardware* e/ou *software* [20] para implementar memória compartilhada. Sistemas implementados em *hardware* utilizam tecnologias de *hardware* especializadas que permitem grande desempenho, porém tornam o sistema complexo e caro. Implementações em *software* possuem o atrativo do baixo custo, contudo tais implementações possuem desempenho limitado devido ao *overhead* imposto pelo protocolo de coerência implementado.

Este capítulo tem por finalidade descrever as características fundamentais de um típico sistema de memória compartilhada distribuída e apresentar a implementação utilizada nos

experimentos realizados. Ele inicia descrevendo as seguintes características: modelo de consistência de memória, protocolo de coerência, granularidade do sistema e localidade dos dados. E finaliza apresentando a implementação em *software* do protocolo HLRC.

2.1 Modelo de Consistência de Memória

O modelo de consistência de memória determina quando os dados modificados por um processador serão visíveis por outros processadores [1].

Diferentes tipos de aplicações paralelas requerem diferentes tipos de modelos de consistência. A escolha do modelo influenciará o desempenho do sistema na execução das aplicações.

Modelos de consistência mais restritos geralmente aumentam a latência dos acessos à memória além de utilizarem mais banda de comunicação, porém simplificam a programação. Para melhorar o desempenho do sistema, uma maneira efetiva é relaxar o modelo de consistência. Um modelo de consistência relaxado permite que a propagação e aplicação das operações de consistência sejam adiadas até um ponto de sincronismo. Isto reduz o impacto ocasionado pelo protocolo implementado pelo sistema, melhorando seu desempenho.

Um modelo de consistência restrito é o modelo de consistência seqüêncial. Este trata os acesso à memória como operações ordinárias de leitura e escrita. Modelos relaxados fazem a distinção entre acessos ordinários e acessos sincronizados. Os modelos denominados consistência fraca [1] e consistência por liberação [1] são variações de modelo relaxado com diferentes graus de relaxamento.

A consistência seqüêncial especifica que todos os processadores do sistema observem a ordem das operações de escrita e leitura na mesma ordem em que seriam realizadas por um único processador.

A consistência fraca faz a distinção entre acessos ordinários e acessos sincronizados.

Isto requer que a memória se torne consistente apenas em acessos sincronizados. Neste modelo, requisições de consistência seqüêncial são aplicadas apenas durante acessos sincronizados. Acessos sincronizados devem esperar que todos acessos anteriores sejam executados, enquanto acessos ordinários devem apenas esperar pelo término do último acesso sincronizado.

Finalmente, a consistência por liberação divide os acessos sincronizados em operações de *acquire* e *release*; portanto, para proteger dados compartilhados, os acessos devem ser realizados entre um par *acquire/release*. Acessos ordinários de leitura e escrita apenas podem ser executados após todos os *acquires* anteriores realizados pelo mesmo processador forem executados. Além disto, um *release* somente pode ser executado após todos os acessos ordinários de leitura e escrita no mesmo processador forem executados.

2.2 Protocolos de Coerência

Um protocolo de coerência de memória especifica quais valores deverão ser visíveis por outros processadores em pontos de sincronização[23]. Um protocolo consiste de uma estrutura de dados e um conjunto de algoritmos que implementam um determinado modelo de consistência. Estes algoritmos possuem duas finalidades: (1) gerenciar as falhas de acesso e (2) implementar as primitivas de sincronização.

O gerenciamento de falhas de acesso determina quais rotinas devem ser executadas quando uma unidade de compartilhamento inválida é acessada. O papel do protocolo mediante uma falha de acesso é buscar uma cópia atualizada do dado referenciado, realizando a consistência da memória.

As primitivas de sincronismo determinam pontos de sincronismo entre os processadores. O protocolo deve garantir que em um ponto de sincronismo todas modificações realizadas por um processador serão vistas pelos outros processadores. Tradicionalmente duas primitivas de sincronismo são fornecidas: (1) *locks* e (2) barreiras. *Locks* delimitam

regiões críticas da aplicação. *Locks* são usados para eliminar possíveis condições de corrida. Barreiras definem pontos de sincronismo entre todos os processadores que compõem o ambiente de execução.

2.3 Granularidade do Sistema

A escolha do tamanho da unidade de compartilhamento afetará o desempenho do sistema [10]. Esta escolha depende de fatores como o custo de comunicação e localidade de referência da aplicação. O custo de comunicação dependerá da rede de interconexão utilizada. Algumas redes locais possuem pouca diferença entre o tempo gasto para enviar uma mensagem de 1-*byte* e uma mensagem de 1024-*bytes*. Para este cenário, utilizar uma granularidade mais grossa gera vantagens. A localidade de referência sugere o uso de uma granularidade fina, para diminuir o potencial de *trashing*. Este pode ocorrer quando duas máquinas acessam diferentes partes de um mesmo bloco, efeito que também é conhecido como falso compartilhamento.

Muitos sistemas utilizam um tamanho fixo de bloco, ao invés de variar a granularidade de acordo com o tamanho do dado compartilhado. Geralmente a granularidade é imposta pelo *hardware* de gerenciamento de memória, pois muitos sistemas de memória compartilhada distribuída utilizam os mecanismos de memória virtual [14] para gerenciar os acessos aos dados compartilhados.

2.4 Localidade dos Dados

A solução mais comum para o problema da localidade dos dados é atribuir uma residência para cada unidade de compartilhamento de dados, por exemplo a página. Cada residência possuirá uma cópia da unidade para escrita (ou uma cópia para leitura) e uma lista de todos os processadores que possuem uma réplica da unidade. Estabelecer a residência fornece um mecanismo simples para a gerência da consistência da memória.

Se no sistema uma unidade de compartilhamento possuir apenas uma cópia para escrita e dois processadores realizarem uma escrita, ocorrerá o problema de sincronismo na escrita. Este pode ser resolvido utilizando um dos seguintes mecanismos [24]: (1) *write-broadcast* ou (2) invalidação.

O mecanismo *write-broadcast* envia, após uma operação de escrita, uma cópia atualizada da unidade de compartilhamento para todos os processadores que possuem uma réplica da unidade. Ao invés de enviar uma cópia atualizada, o mecanismo de invalidação envia uma notificação para cada processador que possuir uma réplica. Ao receber uma notificação de invalidação, a réplica local é invalidada. Neste mecanismo, um acesso à uma réplica inválida gera uma falha de acesso e uma cópia atualizada é requisitada à residência.

A seguir duas técnicas para a escolha de residência de uma unidade de compartilhamento são descritas.

Residência Fixa. Nesta técnica, durante toda a execução da aplicação, uma unidade pertence a uma mesma residência. O proprietário pode ser atribuído de duas maneiras: (1) ao iniciar o sistema as unidades de compartilhamento são distribuídos pelos processadores que compõem o ambiente de execução ou (2) durante a execução da aplicação o primeiro processador a acessar a unidade se torna a residência.

Residência Dinâmica. A técnica de residência dinâmica determina que a residência de uma unidade pode variar durante a execução da aplicação. A atribuição de uma residência pode ocorrer de forma similar a técnica descrita anteriormente, contudo a residência poderá ser alterada à medida que as escritas em uma determinada unidade se concentrem em outro processador diferente.

2.5 A Implementação do Protocolo HLRC

O protocolo *home-based lazy release consistency* [7] é implementado utilizando uma variação do protocolo de consistência por liberação preguiçosa [11] através de um *software* para detectar escritas e um esquema de propagação de escritas baseado em mecanismo de *diffs*, que são registros contendo todas as modificações das página associadas. HLRC utiliza um nó residência para cada unidade de compartilhamento, no qual atualizações são coletadas e aplicadas, garantindo que em seu nó residência a unidade de compartilhada esteja sempre atualizada.

A granularidade da unidade de compartilhamento é o tamanho de uma página utilizada pelo sistema operacional Linux, possibilitando usar chamadas ao sistema para gerenciar as falhas de acessos. Acessos à uma unidade de compartilhamento inválida são satisfeitos solicitando uma cópia da unidade à sua residência.

A execução de cada processo é delimitada em intervalos identificados por um índice. Um intervalo é o tempo entre duas operações de sincronização consecutivas visto por um único processo [22]. Um novo intervalo é iniciado quando um processo executa uma operação de sincronismo. Intervalos entre processadores são parcialmente ordenados. Um intervalo do processador A precede um intervalo do processador B se o intervalo de B inicia com um *acquire* correspondente a um *release* concluído pelo intervalo de A. Esta ordem parcial pode ser representada por um vetor *time-stamp* para cada intervalo. Intervalos são mantidos por processadores e não por processos. Durante o intervalo corrente, cada processador mantém uma lista chamada *upadate-list* que contém referências de todas as páginas modificadas. Quanto um processo termina seu intervalo, a *update-list* é colocada em uma estrutura chamada *bins*. Um *bins* informa quais páginas foram modificadas em relação as operações de sincronização e em quais ordens, mostrando quais páginas devem ser invalidadas durante uma operação de sincronização. Esta informação é também usada para inferir informações sobre as versões das páginas, a medida que páginas são

buscadas do nó *home*.

Em HLRC, *diffs* são computados no final de cada intervalo para todas as páginas modificadas. Após criados, eles são enviados aos seus respectivos *homes* onde são imediatamente aplicados. Escritores podem discartar *diffs* logo após enviá-los. *Homes* de páginas aplicam *diffs* a medida que estes chegam e em seguida também discarta-os.

Para a sincronização, a interface disponibilizada pelo sistema fornece ao programador primitivas de *locks* e barreiras.

Cada *lock* possui um gerente, que é atribuído estaticamente, utilizando a política *round-robin*. O gerente mantém um histórico dos processadores que requisitaram o seu *lock*. Quando um processador executa uma operação *acquire*, este envia ao processador que executou o *release* correspondente um vetor que indica quais *bins* estão presentes em seu processador. A porção do *bins* que não existir no processador que realizou o *acquire* retorna na forma de *write-notices*. Finalmente, o processador que fez a requisição do *lock* aplica os *write-notices*, invalidando as páginas referenciadas.

Barreiras são implementadas usando um gerente centralizado. Cada processador ao chegar em uma barreira atualiza seus *bins* e os envia ao gerente da barreira. O gerente então propaga estas informações para todos os nós do sistema. Ao receber as informações do gerente, cada processador atualiza seus *bins* locais, invalidando todas as páginas necessárias.

Capítulo 3

cJava

cJava pretende prover os desenvolvedores com um máquina virtual Java que esconde a complexidade da programação em *clusters*, fornecendo a possibilidade de programas *multithread* serem executados em uma plataforma com recursos fisicamente distribuídos.

cJava fornece a aplicação Java uma camada de abstração que encapsula os recursos do *clusters*. Este encapsulamento fornece à aplicação a ilusão de estar sendo executada em um simples computador multiprocessado.

O ambiente de execução é composto por um conjunto de máquina interconectadas por uma rede de alta velocidade, e cada máquina do *clusters* executa uma instância de uma máquina virtual Java modificada. Cada instância possui a capacidade de cooperar com outras instâncias e executar um conjunto de *threads*.

Aplicações *multithread* podem ser executadas por cJava sem nenhuma alteração, devido a implementação distribuída ser baseada através de modificações na máquina virtual Java e não no fornecimento de novos mecanismos de programação. A abordagem utilizada visa manter a portabilidade das aplicações *multithread* Java existentes.

Este capítulo descreve as estratégias utilizadas para implementar um ambiente de execução no qual as *threads* da aplicação são distribuídas automaticamente pelas máquinas do *clusters*. A próxima seção descreve a arquitetura do ambiente de execução, para as posteriores descrevem os subsistemas que compõem a arquitetura e a última descrever a implementação do ambiente.

3.1 A Arquitetura do Ambiente

A figura 3.1 demonstra a arquitetura do ambiente de execução cJava. No topo está localizada a aplicação do usuário, capaz de criar diversas *threads* que acessam um espaço global de objetos. Estas camadas são gerenciadas através dos serviços de três importantes subsistemas que gerenciam memória compartilhada distribuída, criação remota de *threads* e comunicação entre as instâncias das máquinas virtuais Java utilizadas por cJava. A implementação destes subsistemas gera um processo de usuário que é executado pelas diferentes máquinas do *clusters*, no topo do sistema operacional Linux.

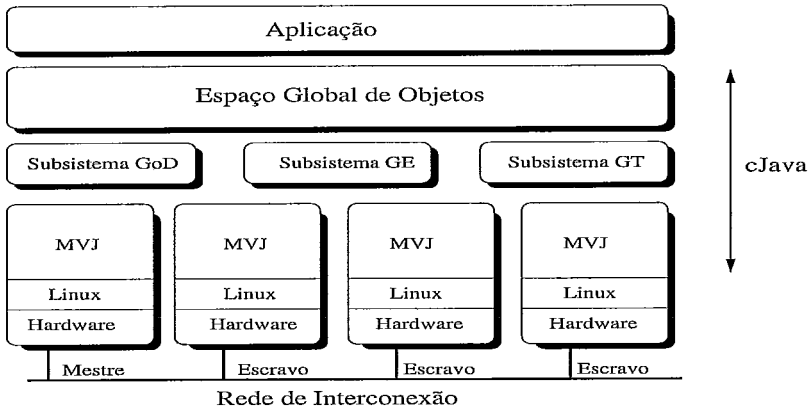


Figura 3.1: Arquitetura do ambiente de execução cJava.

Cada máquina virtual Java utilizada por cJava contêm três subsistemas:

- Subsistema Gerente de Objetos Distribuídos. Subsistema que aloca e gerencia um espaço global de objetos.
- Subsistema Gerente de Eventos. Subsistema responsável por realizar a comunicação entre as instâncias das máquinas virtuais Java.
- Subsistema Gerente de *Threads*. Subsistema capaz de criar *threads* remotas e realizar o sincronismo distribuído, entre as máquinas do *clusters*.

A máquina virtual Java executada em cada máquina é um processo independente capaz de compartilhar um subconjunto de suas estruturas internas. Elas compartilham o

repositório de objetos e suas estruturas de controle. Estes são alocados em um espaço de endereçamento virtual comum entre as máquinas do *clusters*. Este espaço de endereçamento comum é fornecido pelo subsistema gerente de objetos distribuídos e será descrito em mais detalhes na seção 3.2.

As máquinas virtuais Java são classificadas em máquinas mestre e escravas. A máquina mestre é aquela que iniciou a aplicação. Esta possui a responsabilidade de distribuir a aplicação pelas máquinas que compõem o *clusters* e enviar um sinal para as máquinas escravas indicando o término da aplicação. As outras máquinas, classificadas como escravas, estão subordinadas a máquina mestre servindo suas requisições. Após receber uma requisição, a máquina escrava irá realizar as operações necessárias e em seguida aguardar por novas requisições.

3.2 Subsistema Gerente de Objetos Distribuídos

O subsistema gerente de objetos distribuídos (GoD) provê acessos aos objetos, independente de localidade. Cada máquina virtual Java possui um subsistema GoD responsável por gerenciar os recursos de memória local e cooperar com outros GoD's executados por outras máquinas virtuais Java, criando um espaço global de objetos. Isto é realizado implementando o GoD no topo de um sistema de memória compartilhada distribuída. Com o auxílio de um sistema de memória compartilhada distribuída, regiões de memória entre as diversas máquinas são unificadas para formar um contíguo espaço de memória para compartilhar objetos. Desta maneira, a localidade onde o objeto reside é transparente para cada *thread*.

O espaço global de objetos é usado para conter todos os objetos Java. Os objetos que são alocados em uma máquina são armazenados pelo sistema de memória compartilhada distribuída. cJava utiliza o protocolo de coerência provido pelo sistema para manter a consistência dos dados. Este espaço global é estabelecido alocando uma grande região de

memória compartilhada.

A vantagem dos sistemas com memória compartilhada distribuída é que todas as máquinas podem acessar o mesmo espaço de objetos e endereços podem ser usados como uma única referência global para um determinado objeto. Estes sistemas resolvem os acessos aos dados realizando um acesso a uma cópia local de memória ou através de um acesso remoto. Este mecanismo elimina a tradução de endereços virtuais quando referencias são trocadas entre máquinas.

A escolha de um sistema com memória compartilhada distribuída deve-se basicamente a duas vantagens principais: (1) a facilidade da utilização do modelo, simplificando a implementação do ambiente de execução e (2) a consistência da área compartilhada ser gerenciada pelo sistema de memória compartilhada, eliminando maiores alterações do código padrão da máquina virtual Java.

O subsistema GoD possui uma *interface* para suportar o modelo adotado. O uso desta *interface* possui o atrativo de esconder da máquina virtual Java o sistema de memória compartilhada distribuída utilizado.

cJava emprega uma abordagem descentralizada para gerenciar a memória, onde cada GoD de diferentes máquinas virtuais é responsável por gerenciar sua própria área compartilhada de memória global. Acessos realizados por uma *thread* a um determinado objeto são sempre satisfeitos localmente. Devido a possibilidade de várias *threads* atualizarem um determinado objeto ao mesmo tempo, as primitivas *lock()* e *unlock()*, fornecidas pelo sistema de memória compartilhada distribuída, são utilizadas para assegurar a integridade dos dados.

A máquina virtual Java interage com o subsistema GoD durante os acessos ao espaço global de objetos ou durante a invocação de uma chamada sincronizada.

3.3 Subsistema Gerente de Eventos

O subsistema gerente de eventos (GE) fornece suporte à gerência de requisições e sinalização remota.

As máquina virtuais Java compartilham uma caixa de mensagens contendo uma fila para cada máquina virtual Java. As filas são utilizadas como depósito de requisições provenientes de máquinas remotas. Durante o tempo de vida da máquina virtual Java, o subsistema GE inspeciona sua fila de requisições verificando se chegaram novas requisições. Ao receber uma requisição, esta é retirada da fila e as ações apropriadas são executadas dependendo do seu tipo.

As requisições gerenciadas pelo subsistema GE compreendem:

- Criar *thread.* Ao receber uma requisição deste tipo, o novo objeto é criado e o método *init* invocado para realizar as inicializações necessárias.
- Iniciar *thread.* Para uma requisição de início de *thread*, o método *start* é invocado iniciando a nova *thread.*
- Terminar aplicação. Quando uma requisição deste tipo é retirada da fila, a máquina virtual Java executa as rotinas padrões para o término da execução.
- Sinalizar. Requisição utilizada para sinalização remota, descrita a seguir.

Para a sinalização remota, o subsistema é capaz de receber e enviar mensagens para outras máquinas virtuais com a finalidade de realizar a comunicação entre as *threads*. Quando uma *thread* invoca o método *notify* (ou *notifyall*), o subsistema envia um sinal de forma tranparente à máquina virtual Java que possui a *thread* que executou um método *wait* sobre o mesmo objeto. Ao receber o sinal, a *thread* é recolocada na fila de pronto para posterior execução. Esta abordagem fornece à aplicação a ilusão de estar executando tais métodos localmente.

3.4 Subsistema Gerente de *Threads*

No ambiente de execução cJava, *threads* podem ser criadas localmente ou em uma máquina remota. Porém a localidade de cada *thread* é transparente para o programador da aplicação.

O subsistema gerente de *thread* (GT) mantêm uma fila contendo os nomes das máquinas do *clusters* que irão participar da computação. Quando um novo objeto *thread* é criado, o método *new* é interceptado e o nome de uma máquina retirada da fila. Se a máquina local for a escolhida, o novo objeto é criado automaticamente. Na escolha de uma máquina remota, uma requisição é enviada a esta máquina contendo uma referência ao objeto e um identificador para determinar o tipo da requisição. Ao receber a requisição, a máquina remota executa as operações necessárias para criar a nova *thread.*

No momento em que uma nova *thread* é criada, uma estrutura de dados é preenchida contendo o identificador do objeto e o nome da máquina retirada da fila. Esta estrutura fornece a informação da localidade de cada objeto *thread.*

A inicialização de uma *thread* ocorre de maneira similar a sua criação. Durante a invocação do método *start*, este é interceptado e a estrutura de dados contendo a localidade das *threads* consultado. Se a máquina que instanciou for a local o método é invocado automaticamente, enquanto que se a máquina for remota uma requisição é enviada para que esta invoque o método *start* da *thread* a ser iniciada.

Uma característica distinta de computação *multithread* é que *threads* podem compartilhar recursos e serem executadas concorrentemente para a multiplexar computação. Em um ambiente de execução *multithread*, um mecanismo para prover exclusão mútua é necessário para coordenar os acessos aos recursos compartilhados de forma a manter a integridade dos dados. A linguagem de programação Java usa monitores para implementar exclusão mútua. Um monitor é associado com cada objeto compartilhado de forma que a aplicação possa evitar qualquer condição de corrida, aguardando pelo uso do monitor

associado antes de atualizar um objeto compartilhado. Sincronismo entre *threads* é uma conseqüência direta do uso de monitores.

O subsistema GT utiliza uma abordagem descentralizada para implementar monitores globais, que são utilizados para gerenciar o bloqueio e reinício das *threads* ativas. Um monitor global é criado na primeira vez que um *monitor enter* é aplicado sobre um objeto. Cada monitor global mantém um contador e uma fila de espera contendo as *threads* bloqueadas aguardando a liberação de um determinado objeto. Se o contador for zero, a *thread* imediatamente bloqueia o objeto e incrementa o contador, caso contrário a *thread* é colocada na fila de espera. Ao liberar um objeto, a *thread* decrementa o contador. Com o contador em zero, a primeira *thread* bloqueada é retirada da fila de espera e recolocada na fila de pronto para posterior execução.

Uma *thread* em execução será bloqueada (1) na tentativa de executar um *monitor enter* sobre um objeto bloqueado ou (2) durante a invocação do método *wait* de um determinado objeto. Uma *thread* bloqueada é recolocada na fila de pronto quando (1) o monitor que ela requisitou é liberado ou (2) outra *thread* ter invocado o método *notify* ou *notifyall* sobre o objeto que a *thread* bloqueada está aguardando.

Como um projeto similar a monitores globais, cJava implementa um mecanismo de sinalização remota entre *threads*, onde qualquer máquina virtual Java pode redirecionar um sinal de *wait* e/ou *notify* (*notifyall*) para um outra máquina virtual. Com monitores globais e um subsistema de sinalização remota, cJava implementa sincronização distribuída entre *threads* usando um mecanismo distribuído. O subsistema de sinalização remota é descrito em detalhes na próxima seção.

3.5 Implementação

As características da linguagem Java, como o fornecimento de classes *thread* e mecanismo para sinalização e sincronismo entre *threads*, simplificaram a implementação do

protótipo.

Algumas modificações foram realizadas na máquina virtual Java para fornecer à aplicação a ilusão de estar sendo executada por um simples computador multiprocessado. Um gerente de objetos distribuídos foi incorporado ao subsistema gerente de memória para criar um espaço global de objetos. Os acessos sincronizados ao espaço global de objetos foram estendidos para utilizar as primitivas *lock()* e *unlock()* fornecidas pelo sistema de memória compartilhada distribuída. Também, o subsistema de *threads* foi estendido para suportar criação remota e monitores globais. Finalmente, um subsistema responsável pela sinalização remota foi incorporado à máquina virtual Java. Estas modificações foram realizadas na versão 1.2 por esta ser a versão disponível no início do projeto.

Como as *threads* fazem acessos à um mesmo repositório de objetos, cJava aloca a área de métodos, a *heap* e suas estruturas de controle na memória compartilhada distribuída. Desta forma, se mantêm a semântica da máquina virtual Java determinando que quando duas *threads* tentam encontrar uma classe que ainda não foi carregada, apenas uma carrega a classe. Para manter a integridade dos dados alocados na memória compartilhada, cJava usa as primitivas *lock()* e *unlock()* fornecidas pelo sistema de memória compartilhada distribuída. O uso destas primitivas foi reduzido ao máximo: elas somente são usadas onde o código da máquina virtual Java prevê condição de corrida.

A especificação da arquitetura do ambiente de execução foi definida de forma a garantir que o protótipo fosse capaz de utilizar qualquer sistema de memória compartilhada distribuída baseado em *software*. Visando que o protótipo tivesse esta característica, a máquina virtual Java utiliza uma *interface* para realizar chamadas ao sistema de memória compartilhada distribuída. O uso de uma *interface* padrão, que esconde da máquina virtual Java o sistema utilizado, proporciona que diferentes sistemas sejam utilizados pelo protótipo. Embora, os experimentos a serem realizados utilizem uma implementação do protocolo HLRC [21], outra implementação poderia ser utilizada, por exemplo o sistema

TreadMarks [12]. As únicas restrições no uso de outro sistema são: (1) que este seja compatível com a *interface* padrão fornecida por cJava e (2) seja implementado através de *software* como uma biblioteca dinâmica. A próxima seção descreve a *interface* padrão fornecida por cJava.

Uma limitação do protótipo atual é que a quantidade de *threads* da aplicação deve ser igual a quantidade de máquinas do *cluster*. Esta limitação é devida a dois fatores: (1) uma limitação da implementação, que não suporta mais de uma *thread* da aplicação por máquina virtual Java e (2) uma limitação da implementação do protocolo HLRC, que também não suporta mais de uma *thread* por máquina do *cluster*.

3.5.1 Interface

A seguir descrevemos cada função disponibilizada pela interface.

InitializeDSM(x, y) inicializa sistema de memória compartilhada distribuída, tendo em **x** a quantidade de parâmetros e em **y** a lista de parâmetros passados ao sistema.

FinalizeDSM() finaliza sistema de memória compartilhada distribuída.

LockAcquireDSM(x) executa uma operação *acquire* para o *lock* **x**.

LockReleaseDSM(x) executa uma operação *release* para o *lock* **x**.

MallocDSM(x) aloca uma região de memória compartilhada de **x** *bytes*.

FreeDSM(x) libera a região de memória compartilhada referenciada por **x**.

BarrierDSM(x) executa uma operação de barreira número **x**.

ResetDSM() inicializa os contadores das estatísticas com o valor zero.

ClockStopDSM() para de coletar as estatísticas.

IdDSM retorna o identificador do processo.

Capítulo 4

Avaliação de Desempenho

O ambiente cJava foi avaliado em uma plataforma Linux de oito PCs Pentium III conectados por um *switch Giganet* cLan 5300. Cada PC é equipado com um processador Intel de 650 MHz, 512 MBytes de memória principal e executa a versão 2.2.14 do *kernel* Linux. A implementação de cJava é baseada na máquina virtual Java da Sun, versão 1.2, e utiliza a implementação por *software* do protocolo HLRC como sistema de memória compartilhada distribuída. A versão do sistema de memória compartilha distribuída utilizada nos experimentos, possui suporte à *VI Architecture* (VIA) [19], um protocolo de comunição de alto desempenho.

O papel da análise aqui apresentada é avaliar e identificar os pontos da implementação que precisam ser otimizados para melhorar o desempenho do ambiente e descrever o impacto do sistema de memória compartilhada distribuída utilizado pelo subsistema gerente de objetos distribuídos.

Os tempos de execução apresentados na análise não incluem o tempo de inicialização seqüêncial, tais como: tempo gasto durante a criação remota de *threads* e tempo gasto para inicializar os elementos de uma matriz ou para carregar dados de um arquivo. Um relógio foi inicializado após a fase de inicialização seqüêncial com a finalidade de termos uma estimativa mais precisa do desempenho.

Este capítulo apresenta a análise de desempenho do protótipo implementado. Ele inicia descrevendo as aplicações utilizadas como *benchmarks*, para posteriormente des-

crever os experimentos realizados.

4.1 Aplicações

O pacote cJava de *benchmarks* é composto por 6 aplicações científicas: Crypt, do *benchmark multithread* disponibilizado pelo *Java Grande Forum* [18]; SOR, LU, FFT, Radix, portadas do *benchmark SPLASH* [29] e MM. A seguir fornecemos uma descrição de cada aplicação.

MM realiza a multiplicação de duas matrizes quadradas de dimensão *D*. Esta aplicação é totalmente paralela não necessitando de sincronismo entre as *threads*. O núcleo da aplicação compreende três laços responsáveis por multiplicar as matrizes origens e armazenar o resultado na matriz destino. As matrizes origens são divididas em blocos de *D/T* linhas e cada bloco atribuído a uma *thread*. MM possui granularidade grossa (1 página) tanto para leitura como para escritas.

Crypt implementa o algoritmo *IDEA (International Data Encryption Algorithm)* [13] que criptografa e descriptografa um *array* de *N bytes*. Este algoritmo envolve dois laços principais, cujas interações são independentes. O *array* de *N bytes* é dividido em *N/T* blocos e cada bloco é atribuído a uma *thread*. Como MM, Crypt é uma aplicação totalmente paralela que possui granularidade de leitura e escrita grossa.

SOR é um método para resolver equações diferenciais. Na versão paralela, o programa divide a matriz em dois *arrays*, *red* e *black*. Estes *arrays* são divididos em faixas iguais de linhas e cada faixa atribuída a uma *thread*. Comunicações ocorrem sobre as linhas entre faixas. O padrão desta aplicação é um produtor e um consumidor. Leituras possuem granularidade fina, enquanto escritas possuem granularidade grossa devido a cada *thread* escrever em todo o bloco a ela atribuído. SOR utiliza barreiras para o sincronismo entre as *threads* da aplicação.

FFT é uma aplicação que calcula a transformada de Fourier para um conjunto de dados complexos de uma arbitrária entrada de dados. Em FFT a comunicação para uma execução com N-pontos é essencialmente uma matriz de transposição de tamanho $\sqrt{N}$ por uma de tamanho $\sqrt{N}$. As matrizes são distribuídas pelas *threads* (T) em contíguos conjuntos de N/T linhas, e as matrizes origem e destino são reservadas para cada transposição. Cada *thread* durante uma transposição lê $\sqrt{N}/P$ por $\sqrt{N}/P$ submatriz de cada outra *thread* e escreve em seu conjunto de linhas. Acessos de escrita possuem granularidade grossa enquanto a granularidade dos acessos de leitura depende de N e T. O padrão da aplicação é um produtor e um consumidor. FFT utiliza barreiras para o sincronismo entre as *threads* da aplicação.

LU é uma aplicação que fatora uma matriz densa. Para possibilitar a localidade de dados e evitar o falso compartilhamento, a aplicação usa uma estrutura de dados otimizada para a matriz. A matriz é dividida em blocos que são atribuídos estaticamente às *threads*. O padrão de LU é um produtor e um consumidor. Cada bloco de 16 x 16 elementos é exatamente 4k *bytes* (1 página), então a granularidade de leitura e escrita são grossas. LU utiliza barreiras para sincronismo entre as *threads* da aplicação.

Radix ordena um conjunto de N chaves inteiras em ordem ascendente. A fase predominante é a de permutação de chaves. O padrão nesta fase é um produtor e um consumidor, uma *thread* lê *N/T* chaves contíguas do vetor origem e as escreve no vetor destino com um alto espalhamento e irregular permutação. A taxa *radix* (R) determina que as escritas são separadas por *T-1* outros conjuntos e as escritas realizadas para diferentes conjuntos são temporariamente permutadas de uma maneira imprevista. Para uma distribuição uniforme de chaves, uma *thread* escreve em um conjunto contíguo de *N/R*T* chaves no vetor destino. Caso a taxa *N/R*T* seja maior que uma página, escritas possuem granularidade fina para média enquanto leituras

possuem granularidade grossa. Em Radix o padrão um produtor e um consumidor também é identificado. Radix utiliza barreiras e *locks* para sincronismo entre as *threads* da aplicação.

As aplicações contidas no pacote cJava de *benchmarks* podem ser classificadas em 2 grupos: (1) MM, Crypt, SOR, FFT e LU como aplicações regulares e (2) Radix como aplicação irregular. As aplicações regulares: SOR, FFT e LU possuem a característica de um único escritor, isto é, um dado compartilhado é escrito somente pelo processador ao qual ele foi atribuído. Utilizando uma estrutura de dados apropriada, elas possuem granularidade de página e uma localidade regular de acesso a dados, além de não existir a necessidade de computar *diffs*. Ações do protocolo são usadas apenas para buscar páginas. Porém, a aplicação irregular Radix possue um padrão irregular de acesso aos dados e necessita criar e aplicar *diffs*.

A sincronização através de barreiras foi implementada utilizando chamada de método nativo, disponibilizada pela linguagem Java. Desta forma, quando uma aplicação realiza uma operação de barreira, a máquina virtual Java modificada executa o método barreira disponibilizado pelo sistema de memória compartilhada distribuída. Outra alternativa seria o programador implementar uma classe barreira, utilizando as operações: *wait* e *notify*.

O uso de *locks* por uma aplicação não necessita ser implementado pelo programador, basta usar uma chamada *synchronized*. Isto, devido uma chamada *synchronized* fazer o uso das primitivas *lock()* e *unlock()* disponibilizadas pelo sistema de memória compartilhada distribuída.

4.2 Parâmetros Utilizados na Análise

Nesta seção descreveremos os parâmetros usados na escolha do conjunto de dados utilizados na análise de desempenho e o tamanho das entradas de cada aplicação.

Os dados apresentados na análise indicam a média amostral de um conjunto de dados contendo *5* elementos. Cada elemento de um conjunto foi escolhido de maneira a obter um desvio padrão inferior a 1%. A tabela 4.1 apresenta o tamanho da entrada de dados utilizada por cada aplicação.

Aplicação	Entrada
MM	1000 x 1000
Crypt	5000000 bytes
SOR	2500 x 2500
FFT	4194304 pontos
LU	2048 x 2048
Radix	8388608 chaves

Tabela 4.1: Tamanho da entrada de dados das aplicações.

4.3 Custos das Modificações na Máquina Virtual Java

Nesta seção comparamos o desempenho das aplicações quando executadas pela máquina virtual Java de cJava e a máquina virtual Java original. Para este experimento as aplicações criam uma única *thread.* Os resultados apresentados na tabela 4.2 demonstram que as modificações realizadas no código padrão da máquina virtual Java ocasionaram pouco impacto no desempenho da máquina virtual Java, entre 0 e 2.28 %, para execuções seqüênciais.

Aplicação	JVM 1.2		*Overhead*
	Original	Modificada	
MM	485.77	490.48	0.97 %
Crypt	304.43	311.53	2.28 %
SOR	304.75	306.46	0.56 %
FFT	227.20	228.21	0.44 %
LU	2345.88	2308.62	0 %
Radix	24.27	24.26	0 %

Tabela 4.2: Tempo de execução seqüêncial expresso em segundos.

Para medir o *overhead* imposto pelas modificações, além de obtermos os tempos para execuções seqüências, as aplicações foram instrumentadas utilizando uma biblioteca para

análise de desempenho [28]. Esta biblioteca é capaz de coletar eventos em tempo de execução das aplicações executadas por processadores comerciais. Os eventos coletados foram: quantidade de instruções executadas, quantidade de instruções *load/store*, percentual de *miss* na *cache* de dados nível 1 e ciclos por instrução (CPI). Através destes eventos, veja tabela 4.3, podemos explicar comparativamente o desempenho da máquina virtual Java original e da máquina virtual Java modificada.

Aplicação	Original / Modificada			
	Instruções Executadas (10^9)	% Instruções *Load/Store*	% *Miss Cache* Dados Nível 1	CPI
MM	215.0 / 215.0	74.98 / 71.08	0.01 / 0.01	1.43 / 1.44
Crypt	145.0 / 145.0	79.54 / 81.15	0 / 0.01	1.36 / 1.39
SOR	108.3 / 108.3	75.29 / 73.81	0.19 / 0.23	1.83 / 1.83
FFT	79.4 / 79.4	91.16 / 91.17	0.17 / 0.17	1.85 / 1.85
LU	547.3 / 547.3	86.79 / 90.14	1.23 / 0.61	2.78 / 2.73
Radix	8.8 / 8.8	87.39 / 85.98	0.12 / 0.12	1.76 / 1.76

Tabela 4.3: Estatísticas coletas para as duas máquinas virtuais.

A quantidade de instruções executadas pela duas versões da máquina virtual Java é bem similar, a diferença varia entre 718 a 15997 instruções. Essa diferença se torna desprezível quando comparada com a quantidade total de instruções. O percentual de instruções *load/store*, das duas versões, varia para cada aplicação. Enquanto, esse percentual é maior para MM, SOR e Radix quando executadas pela máquina original, em Crypt, FFT e LU é maior na máquina modificada.

O *overhead* imposto, pelas modificações na máquina virtual Java, às aplicações MM, Crypt e SOR é ocasionado por um ou mais dos seguintes fatores: (1) aumento do percentual de instruções *load/store* e (2) aumento do percentual de *miss* na *cache* de dados.

Para a aplicação MM, a versão modificada possui um maior CPI. Embora, o valor do CPI esteja bem próximo do valor obtido pela versão original, esse valor se torna relevante quando comparado com a quantidade total de instruções executadas. O aumento do valor do CPI indica o aumento da quantidade de ciclos necessários à execução da aplicação. A versão modificada possui 10^{12} ciclos a mais do que a versão original, o que ocasionou o

overhead de 0.97 % (4.71 s).

O *overhead* de 2.28 % da aplicação Crypt é ocasionado por dois fatores: (1) aumento do percentual de instruções *load/store* e (2) aumento do percentual de *miss*.

Na execução da aplicação SOR, apesar da versão modificada possuir um menor percentual de instruções *load/store*, essa versão possui um maior percentual de *miss* na *cache* de dados ocasionando o *overhead* de 0.56 %.

O *overhead* de 0.44 % para a aplicação FFT, quando executada pela máquina modificada, é uma questão ainda aberta após a realização desses experimentos. Isto porque os números para as duas versões são muito próximos.

As modificações na máquina virtual Java não ocasionaram nenhum impacto para a execução das aplicações LU e Radix.

O desempenho similar da aplicação LU é ocasionado pela redução do percentual de *miss* e pela redução do valor do CPI, embora a versão modificada possua um maior percentual de instruções *load/store*.

Durante a execução da aplicação Radix, pela versão modificada, o percentual de instruções *load/store* foi reduzido ocasionando um desempenho similar ao da versão original.

Os resultados sugerem que o *overhead* ocasionado pelas modificações é influenciado pelos parâmetros: (1) percentual de instruções *load/store* e/ou (2) percentual de *miss*.

4.4 Custos da Distribuição da Máquina Virtual Java

Embora as aplicações regulares forneçam localidade regular de acesso a dados e eliminem a necessidade de se computar *diffs*, quando executadas por cJava essas características são alteradas. Em cJava essas aplicações possuem uma localidade irregular de acesso a dados, devido a política de atribuição de residência da implementação HLRC conflitar com a semântica da linguagem Java. A semântica de Java define que um objeto criado deve ter todos os seus campos preenchidos com valores nulos. Como a implementação

HLRC utiliza a política do primeiro toque para atribuir uma residência a uma unidade de compartilhamento, a *thread* que instancia uma nova classe ou um novo *array* passa a ser a sua residência. Isso causa o nó 0 do *cluster* ser sobrecarregado. Este conflito gera uma distribuição desbalanceada dos dados impondo uma busca desnecessária de páginas e computação de *diffs*, além de aumentar o *overhead* imposto pelo sistema de memória compartilhada distribuída.

Para eliminar esse conflito otimizações precisam ser feitas no código da máquina virtual Java ou na política de atribuição de residência do sistema de memória compartilhada distribuída. A vantagem de se alterar a política de atribuição de residência, é manter a semântica da linguagem Java inalterada.

Apesar do aumento do *overhead* imposto pelo sistema de memória compartilhada distribuída, as aplicações obtiveram bons *speedups*, veja tabela 4.4. No ambiente cJava, os *speedups* obtidos para execuções com 8 *threads* variam entre 7.56 à 7.84 para as aplicações totalmente paralelas e entre 6.33 à 6.96 para as aplicações que necessitam de sincronização. Acreditamos que eliminando a localidade irregular de acesso a dados o *overhead* imposto pelo sistema de memória compartilhada distribuída diminuirá, ocasionando uma melhora ainda maior nos *speedups* obtidos.

Aplicação	*Threads*		
	2	4	8
MM	1.98	3.95	7.84
Crypt	1.91	3.83	7.56
SOR	1.83	3.31	6.76
FFT	1.83	3.48	6.11
LU	2.01	3.76	6.96
Radix	1.80	3.45	6.33

Tabela 4.4: *Speedups* obtidos pelas aplicações.

As figuras 4.1 à 4.6, mostram como o tempo de execução é gasto pelas aplicações. Cada figura contém os tempos de execução para 2, 4 e 8 *threads* dividido em dois componentes: (1) tempo de computação útil e (2) tempo gasto pelo sistema de memória com-

partilhada distribuída (MCD).

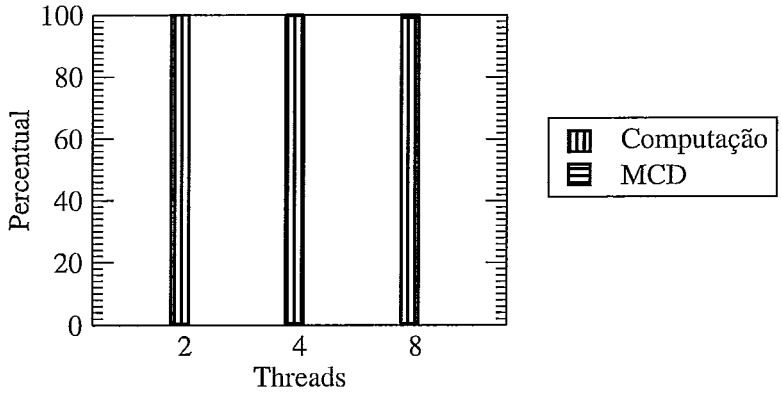


Figura 4.1: Desempenho da aplicação MM.

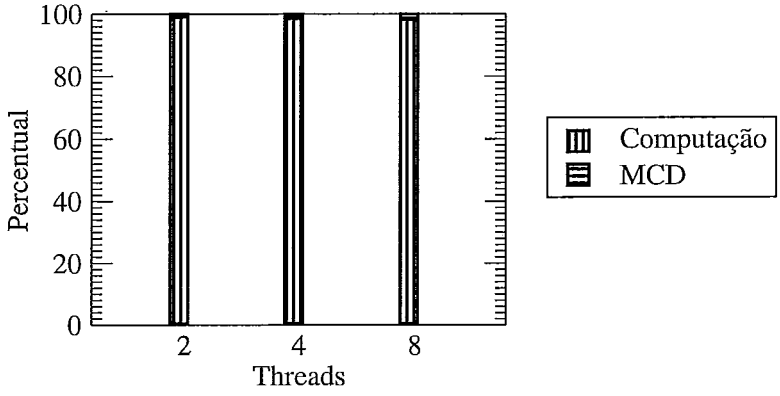


Figura 4.2: Desempenho da aplicação Crypt.

Por não existir comunicação e sincronismo entre as *threads* da aplicação MM (figura 4.1), esta obteve um *speedup* de 7.84 para uma execução com 8 *threads*. MM possui 0.81 % do tempo de execução gasto pelo sistema de memória compartilhada distribuída.

O alto grau de paralelismo também é observado na aplicação Crypt (figura 4.2). Isto explica o *speedup* de 7.56 com 8 *threads*. Apenas 1.78 % do tempo de execução é gasto pelo sistema de memória compartilhada distribuída.

Em SOR (figura 4.3) o *speedup* é degradado pelo protocolo MCD. Para uma execução com 8 *threads* 14.81 % do tempo de execução é gasto pelo sistema de memória compartilhada distribuída, limitando o *speedup* em 6.76.

FFT (figura 4.4) obteve um *speedup* de 6.11 para execuções com 8 *threads*. O tempo

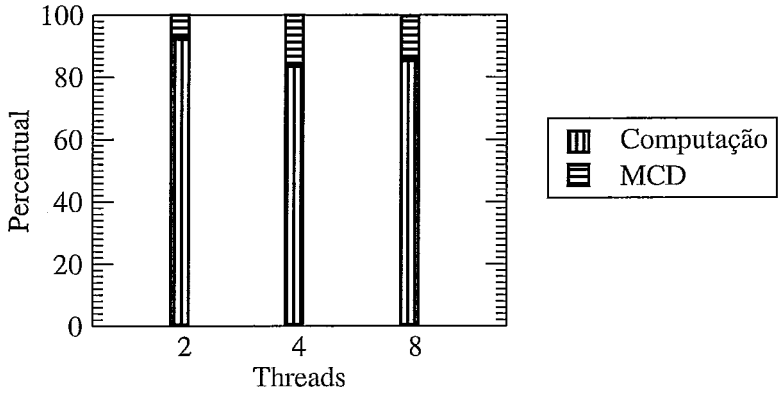


Figura 4.3: Desempenho da aplicação SOR.

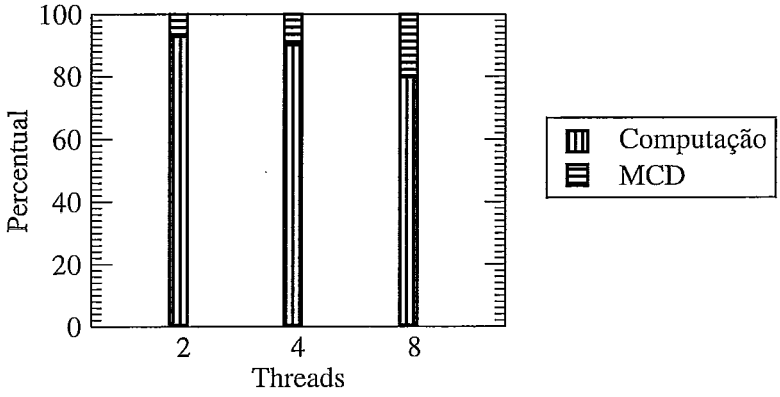


Figura 4.4: Desempenho da aplicação FFT.

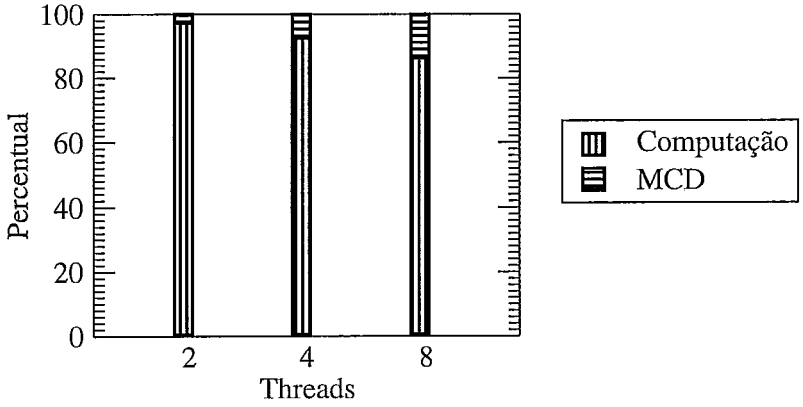


Figura 4.5: Desempenho da aplicação LU.

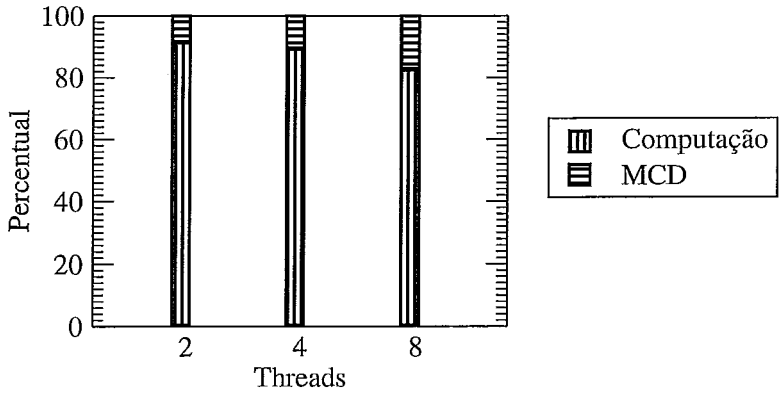


Figura 4.6: Desempenho da aplicação Radix.

gasto pelo sistema de memória compartilhada distribuída é responsável por 20.27 % do tempo de execução.

Para execução com 8 *threads*, a aplicação LU (figura 4.5) obteve o *speedup* de 6.96, limitado por 17.75 % do tempo de execução ser gasto pelo sistema de memória compartilhada distribuída.

O *speedup* de 6.33 obtido pela aplicação Radix (figura 4.6) durante execução com 8 *threads*, foi limitado por 17.61 % do tempo de execução ser gasto pelo sistema de memória compartilhada distribuída.

4.4.1 Análise Detalhada dos Custos

Esta seção apresenta uma análise detalhada do tempo gasto pelo sistema de memória compartilhada distribuída. As figuras 4.7 à 4.12 mostram os componentes do tempo MCD. O tempo MCD é dividido em quatro componentes distintos: paginação, barreira, *lock* e outros. Paginação indica o tempo gasto em falhas de páginas e em busca de páginas de máquinas remotas. *Lock* é o tempo gasto em obter um determinado *lock* de um outro dono. Barreira é o tempo gasto esperando por mensagens de barreira vinda de máquinas remotas. Outros é o tempo gasto realizando ações do protocolo, por exemplo, aplicação de *diffs*.

As aplicações podem ser divididas em três grupos de acordo com o componente de

mais impacto:

1. Aplicações em que o tempo MCD é dominado pelo tempo gasto em paginação: MM,Crypt e FFT.

2. Aplicações em que o tempo MCD é dominado pelo tempo gasto em barreiras: SOR e LU.

3. Aplicações em que o tempo MCD é dominado pelo tempo gasto em paginação e em barreiras: Radix.

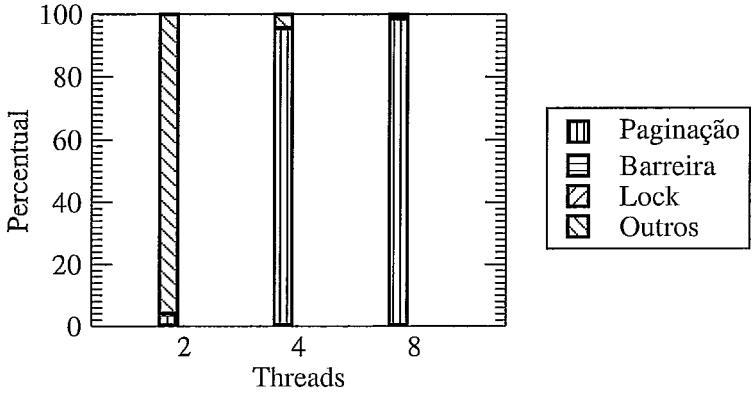


Figura 4.7: Descrição do tempo MCD para a aplicação MM.

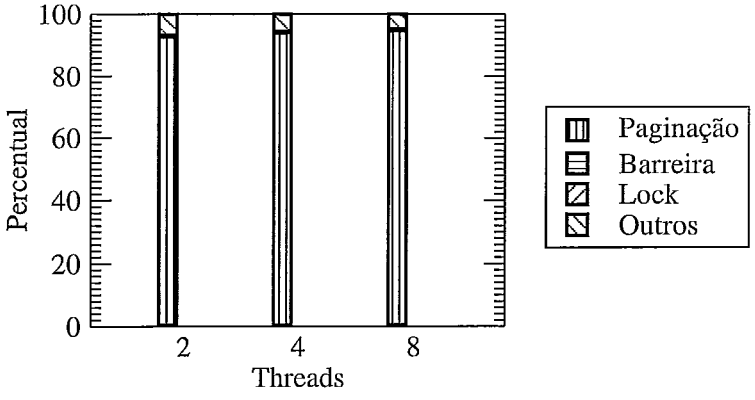


Figura 4.8: Descrição do tempo MCD da aplicação Crypt.

Para uma execução com 8 *threads* das aplicações do primeiro grupo, o percentual do tempo gasto com paginação varia entre 78.25% e 98.52%. FFT é a aplicação com o menor percentual gasto em paginação, 78.25%. Enquanto para MM, o tempo MCD é

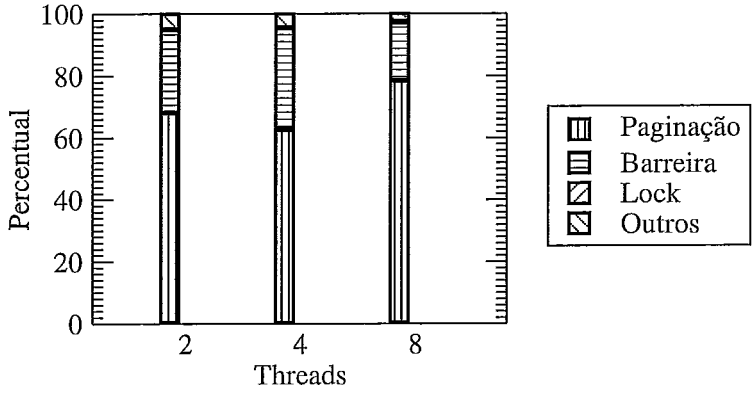


Figura 4.9: Descrição do tempo MCD da aplicação FFT.

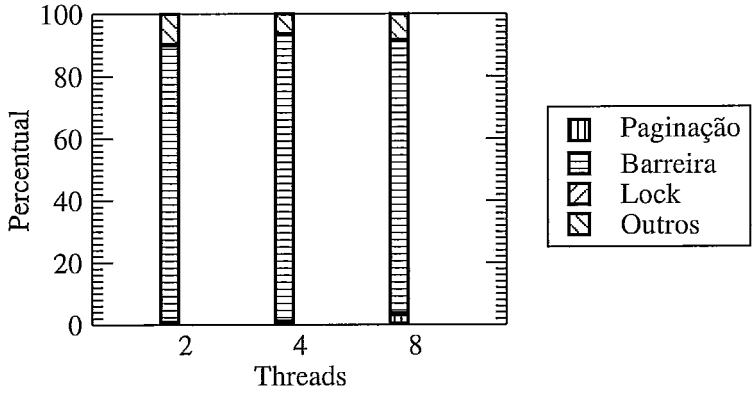


Figura 4.10: Descrição do tempo MCD da aplicação SOR.

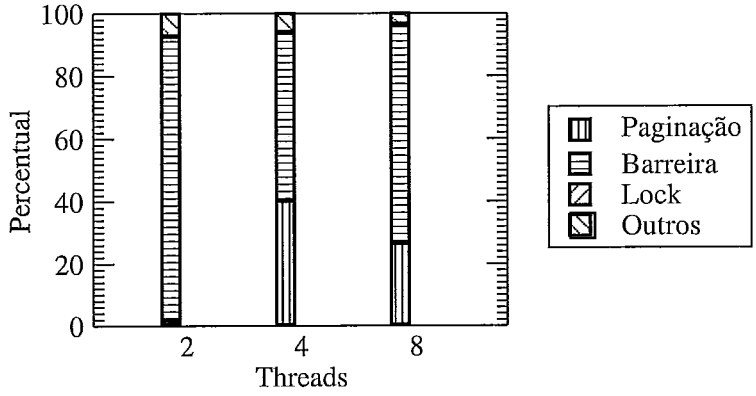


Figura 4.11: Descrição do tempo MCD da aplicação LU.

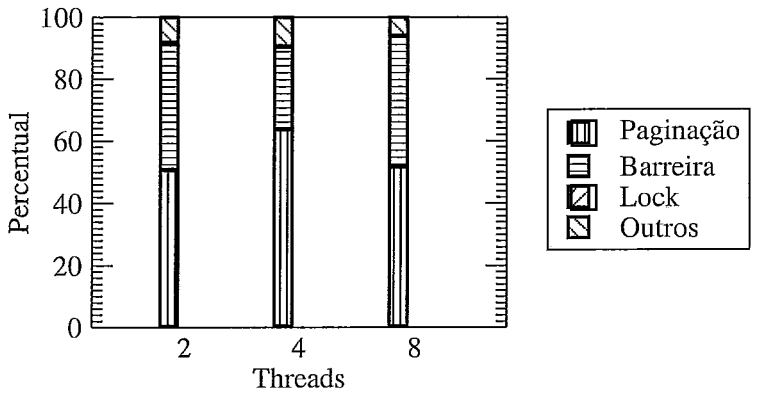


Figura 4.12: Descrição do tempo MCD da aplicação Radix.

totalmente dominado pelo tempo gasto em paginação, 98.52%. Para Crypt o tempo gasto em paginação possui um percentual de 94.54% do tempo MCD.

No segundo grupo de aplicações o percentual do tempo MCD gasto em barreira é de 70.02% para a aplicação LU e 88.11% para a aplicação SOR, isto durante uma execução com 8 *threads*.

Radix é a aplicação que possui um percentual de 51.44% do tempo MCD gasto em paginação e 42.20% do tempo MCD gasto em barreiras.

O uso de *locks* pela máquina virtual Java não influenciou no desempenho das aplicações. O percentual deste componente no tempo MCD está entre 0% e 0.37% para as execuções com 8 *threads*.

Ações do protocolo, componente "outros" do tempo MCD, embora possuam um percentual baixo, entre 1.48% a 8.72% do tempo MCD para execuções com 8 *threads*, acreditamos que possam ser diminuídas eliminando a localidade irregular de acesso a dados.

Os resultados sugerem que a paginação afeta mais do que a barreira. Melhorando a distribuição das residências, além de diminuir os tempos gastos pelos componentes paginação e barreira, eliminará a necessidade de computar *diffs* das aplicações regulares.

4.5 JDK 1.2 Versus JDK 1.4

As aplicações também foram executadas, com uma *thread* apenas, em uma versão mais atualizada, JDK 1.4, distribuída recentemente. As otimizações realizadas em JDK 1.4 ocasionaram significativamente a redução da quantidade de instruções executadas, de 2.19 a 9.57, veja tabela 4.5, reduzindo o tempo de execução seqüêncial das aplicações, veja tabela 4.6.

A aplicação que obteve uma menor taxa de aceleração foi LU, com 2.02. Embora a quantidade de instruções executadas na aplicaçâo LU seja reduzida em 6.98 vezes, a aceleração de apenas 2.02 pode ser explicada pelo CPI de 9.58 obtido pela versão 1.4, veja tabela 4.7, e pelo aumento do percentual de intruções *load/store*, veja tabela 4.7. Enquanto que para LU o CPI aumentou para a versão 1.4, para as outras aplicações essa versão possui um CPI menor. O percentual de intruções *load/store* que varia entre 47.02 % a 94.12 %, também aumentou para a aplicação Radix, contudo, esse percentual foi reduzido para as outras aplicações.

As taxas de aceleração variam dependendo das características de cada aplicação, por exemplo, a aplicação FFT foi a que obteve uma maior taxa de aceleração, 15.26. Essa maior taxa é resultante da redução do CPI e da alta redução do percentual de instruções *load/store*.

Os resultados sugerem que: (1) para aumentar a aceleração de uma aplicação não basta reduzir a quantidade total de instruções executadas, mas também é preciso diminuir o CPI; (2) diminuindo o CPI e eliminando ao máximo instruções *load/store*, obtemos a maior taxa de aceleração, isto é, a aceleração é influenciada pelo CPI e pelo percentual de instruções *load/store*; e (2) o aumento do CPI e do percentual de instruções *load/store* ocasionam a pior taxa de aceleração.

A melhora de desempenho para todas as aplicações justifica a mudança de versão da máquina virtual Java usada em cJava.

A redução do tempo de execução das aplicações levanta a questão de desempenho das aplicações se cJava utilizasse a versão 1.4 da máquina virtual Java, mas que foge ao escopo dessa tese.

Como esperado, a tendência é diminuir o CPI se o percentual de instruções *load/store* diminuir (MM, Crypt, SOR, FFT e LU), exceto Radix. Neste caso, a principal explicação é o aumento na taxa de acerto da *cache* de instruções, que não pode ser medida neste experimento.

Aplicação	Taxa de Redução	Aceleração
MM	5.38	5.75
Crypt	2.19	3.07
SOR	5.70	7.44
FFT	9.57	15.26
LU	6.98	2.02
Radix	5.27	6.39

Tabela 4.5: Taxa de redução da quantidade de instruções e aceleração da versão 1.4 sobre a versão 1.2.

Aplicação	JDK	
	1.2	1.4
MM	485.77	84.42
Crypt	304.43	99
SOR	304.75	40.95
FFT	227.20	14.89
LU	2345.88	1160.53
Radix	24.27	3.79

Tabela 4.6: Tempo de execução seqüêncial expresso em segundos.

Aplicação	JDK 1.2 / JDK 1.4		
	Instruções Executadas (10^9)	% Instruções Load/Store	CPI
MM	215.0 / 40.0	74.98 / 67.18	1.43 / 1.36
Crypt	145.0 / 66.3	79.54 / 71.52	1.36 / 0.96
SOR	108.3 / 18.9	75.29 / 65.16	1.83 / 1.38
FFT	79.4 / 8.3	91.16 / 47.02	1.85 / 1.17
LU	547.3 / 78.4	86.79 / 93.65	2.78 / 9.58
Radix	8.8 / 1.6	87.39 / 94.12	1.76 / 1.45

Tabela 4.7: Estatísticas coletadas para as duas máquinas virtuais.

Capítulo 5

Trabalhos Relacionados

O uso de Java em máquinas distribuídas tem sido estensamente estudado desde que a linguagem Java foi criada. Do ponto de vista dos programadores, várias infraestruturas [17] que fornecem suporte à execução de aplicações Java em *clusters* estão disponíveis, cada uma com a finalidade de fornecer um ambiente Java eficiente para computação de alto desempenho.

Este capítulo descreve trabalhos que usam o mesmo mecanismo utilizado por cJava para realizar a comunicação entre processos. Dos ambientes apresentados, Java/DSM e MultiJav são bem similares quando comparado com a abordagem utilizada na implementação de cJava. Os outros dois ambientes, cJVM e Jackal, embora realizem a comunicação entre processos de forma similar a cJava, utilizam uma abordagem completamente diferente para implementar o ambiente.

5.1 Java/DSM

Java/DSM [30] é um ambiente Java para computação de alto desempenho em *clusters*. Java/DSM fornece suporte à ambientes heterogêneos e utiliza *software* de memória compartilhada distribuída para implementar a comunicação entre processos.

Cada máquina que compõe o ambiente possui uma instância da máquina virtual Java, capaz de compartilhar os objetos criados pela aplicação. Java/DSM utiliza o sistema TreadMarks [2] para fornecer uma região de memória compartilhada, utilizada para conter

a área de métodos e a *heap*.

Para a gerência da memória, cada máquina virtual Java pode realizar a coleta de lixo de maneira independente na maioria das vezes. Porém, para a liberação das regiões de memória utilizadas por estruturas cíclicas, é utilizado sincronismo entre os coletores.

O coletor do lixo de cada máquina mantêm duas listas: a lista de exportação, que contêm as referências remotas para os objetos criados localmente; e a lista de importação, que contêm referências aos objetos remotos. Antes de enviar uma mensagem à outra máquina, o ambiente de execução invoca o coletor de lixo para verificar se a mensagem contêm referências válidas aos objetos locais e inserir as referências válidas na lista de exportação. Quando uma mensagem é recebida, estas são inspecionadas e as referências aos objetos remotos são coletadas e inseridas na lista de importação. O coletor de lixo utiliza um algoritmo contador de referências para decidir quando uma referência pode ser retirada da lista de exportação.

A conversão de dados é requerida para o suporte a plataformas heterogêneas. Java/DSM identifica o tipo do dado para determinar como a conversão deve ser realizada.

Duas restrições são impostas ao programador: (1) uma *thread* não pode migrar entre máquinas, e (2) a localidade de uma *thread* não é transparente. Estas restrições limitam o balanceamento dinâmico de carga e requerem que o programador esteja ciente da localidade de cada *thread.*

Em cJava a criação de *threads* ocorre de forma transparente à aplicação, abordagem que não requer do programador o conhecimento da localidade de cada *thread.* Outra vantagem sobre Java/DSM é a capacidade de utilizar qualquer sistema de memória compartilhada distribuída, que seja compatível com a *interface* fornecida pelo subsistema GoD.

5.2 Multiple Java Virtual Machine

Multiple Java Virtual Machine (MultiJav) [5] é uma implementação distribuída da máquina virtual Java. Cada máquina virtual Java é executada como um processo independente e todas as máquinas que participam da computação são totalmente conectadas por *sockets* [25].

As aplicações paralelas são iniciadas em uma máquina virtual Java denominda de *root*, porém *threads* migram para outras máquinas virtuais Java provendo um sistema com balanceamento de carga.

A carga de classes é realizada localmente, contudo, se uma classe não pode ser encontrada localmente, a máquina *root* fica encarregada de realizar a carga.

Em MultiJav todos os objetos são potencialmente compartilhados, pois o programador não pode declarar explicitamente objetos compartilhados. Porém, o ambiente de execução MultiJav pode detectar automaticamente um objeto que deve ser compartilhado. A identificação deste objeto é obtida através da análise das instruções *load/store* do *bytecode* que está sendo executado.

Para realizar a gerência da sincronização em um ambiente distribuído, MultiJav implementa monitores globais. Cada monitor criado é global a todas as máquinas e operações em uma determinada máquina são visíveis a todas. Cada monitor em MultiJav possui uma máquina como sua residência. Cada máquina possui uma fila associada a cada monitor, que contém *threads* locais e requisições de *threads* (que representam *threads* em máquinas remotas que solicitaram o monitor). Durante as operações de *monitor enter* e *monitor exit*, se a *thread* estiver localizada na residência do monitor ela imediatamente executa a operação, senão ela envia uma requisição à residência do monitor e fica bloqueada aguardando obter o monitor.

MultiJav implementa *handles* globais como identificador de cada objeto. Cada máquina mantém uma tabela de *handles* globais, que mapeiam *handles* globais em *handles*

locais. Ao acessar um objeto, a tabela de *handles* globais é consultada para determinar a máquina virtual Java onde o objeto está localizado e seu respectivo *handle* local.

Múltiplas cópias de um objeto compartilhado podem existir entre as máquinas virtuais Java. Para evitar o falso compartilhamento, *threads* podem ler e escrever no mesmo objeto simultaneamente através do uso de um protocolo com múltiplos escritores.

MultiJav implementa o modelo de consistência por liberação utilizando um mecanismo de *diffs* para enviar as atualizações em pontos de sincronismo, que ocorrem durante as operações de *monitor enter* e *monitor exit*.

Como Java/DSM, MultiJav não apresenta nenhuma análise de desempenho. cJava e MultiJav introduzem o modelo de memória compartilhada distribuída modificando a máquina virtual Java. MultiJav difere de cJava por realizar uma análise para identificar objetos compartilhados e utilizar um *handle global* como identificador de cada objeto. Em cJava todos os objetos são compartilhados devido o uso de uma única *heap*, que é compartilhada entre as *threads* da aplicação. Este mecanismo também dispensa o uso de um *handle global* como identificador para cada objeto.

5.3 Compliant Java Virtual Machine

Compliant Java Virtual Machine (cJVM) [3] é uma implementação baseada em *cluster*. cJVM virtualiza o *cluster*, fornecendo a aplicação um sistema com única imagem.

Vários processos cJVM coexistem simultaneamente no *cluster*, cada processo é uma instância da máquina virtual Java capaz de interpretar *bytecodes* e gerenciar uma porção dos objetos criados pela aplicação.

cJVM distribui os objetos e *threads* entre as máquinas virtuais Java visando obter escalabilidade. A criação de *threads* remotas é implementada estendendo o modelo de *threads* padrão da máquina virtual Java. Para as classes que implementam a interface *Runnable*, cJVM reescreve o *bytecode new* transformando-o em *remote_new*. Quando

executado, *remote_new* usa uma função de balanceamento de carga para determinar a melhor máquina onde a *thread* deverá ser criada.

Para suportar acessos aos objetos distribuídos, cJVM implementa o modelo *master-proxy*. Neste modelo, a cópia *master* de um objeto é armazenada na máquina onde este foi criado. Para que as outras máquinas acessem este objeto, utiliza-se um *proxy*. O mecanismo *master-proxy* é suportado pela alteração do modelo padrão de objetos Java. cJVM adiciona ao modelo padrão de objetos, tabelas contendo funções para um número pré-definido de *proxies*.

Objetos são alocados na *heap* local e referenciados localmente pela máquina virtual Java. Um identificador global é criado para cada objeto que é passado como argumento de um método remoto. Este identificador é utilizado para localizar a cópia *master* de um objeto *proxy*. Quando um objeto *proxy* é acessado, as chamadas de métodos são redirecionadas de maneira transparente para a máquina virtual Java onde a cópia *master* está localizada.

cJVM realiza a gerência dos acessos aos objetos alterando os *bytecodes* que fazem acessos à *heap*. A alteração destes *bytecodes* tem a finalidade de determinar se os dados a serem acessados são locais ou remotos.

cJVM difere de cJava por empregar uma abordagem diferente para implementar memória compartilhada distribuída. O uso do modelo de objetos *master-proxies* utilizado por cJVM possui a desvantagem de ocasionar contenção na rede se um objeto é acessado freqüentemente por várias máquinas, o que não ocorre com cJava. cJVM possui o atrativo de fornecer ao programador mecanismos para controlar a carga do sistema. cJava possui a desvantagem de não fornecer ao programador nenhum mecanismo com esta finalidade.

5.4 Jackal

Jackal [26] implementa uma abstração de memória compartilhada distribuída através de um ambiente de execução e um compilador. O compilador traduz código Java em código executável, ao invés de *bytecodes*, e é capaz de realizar busca adiantada de dados.

O protocolo de coerência implementado por Jakal é baseado em invalidações e em residência de regiões. Regiões são definidas como objetos ou partições de *arrays*. Este protocolo determina que, em pontos de sincronismo, cada *thread* invalide seus próprios dados para assegurar a consistência nos próximos acessos. Embora tal protocolo seja simples, ele pode invalidar dados que não serão acessados por nenhuma outra *thread*, acarretando um *overhead* adicional ao mecanismo de coerência. Para implementar o protocolo de invalidação, cada *thread* mantêm uma lista de controle das regiões acessadas para leitura e escrita desde o último ponto de sincronismo. Em pontos de sincronismo, as cópias presentes na *cache* que possuem referências na lista são invalidadas e as regiões modificadas são enviadas a suas localizações originais.

Para evitar a tradução de endereço desnecessária, as regiões alocadas em diferentes máquinas possuem o mesmo endereço virtual. O compilador gera uma validação de acesso para cada objeto ou *array*. Esta verificação determina se a região referenciada contêm uma cópia válida. Ao detectar um acesso inválido, o ambiente de execução solicita à residência da região uma cópia atualizada.

Jackal fornece um modelo de memória diferente do padrão Java e possui as seguintes premissas: (a) um programa não possui condições de corrida, (b) um programa possui declarações de sincronismo suficientes, e (c) estas declarações são aplicadas a objetos e *arrays*.

Jackal implementa coleta de lixo local e global. Quando um nó não possui memória livre, ele executa uma coleta de lixo local. Para a coleta local nenhum sincronismo é necessário. Entretanto, o coletor de lixo local não pode discartar os objetos que fazem

referências à objetos remotos. Quando a quantidade de referências à objetos remotos se torna suficientemente grande, a fase global do coletor é inicializada.

cJava utiliza uma abordagem complementamente diferente da utilizada por Jackal para implementar uma abstração de memória compartilhada distribuída. Além desta diferença, a máquina virtual Java utilizada por cJava interpreta *bytecodes*, enquanto Jackal utiliza um compilador para traduzir código Java em código nativo.

Capítulo 6

Conclusões

Nesta tese, propusemos e implementamos cJava, uma plataforma Java para computação de alto desempenho em *clusters*. cJava possui suporte à criação remota de *threads*, acesso a um espaço global de objetos e sincronização distribuída para alcançar a ilusão de uma máquina multiprocessada.

O sistema de memória compartilhada distribuída que implementamos utiliza consistência por liberação assumida por Java e demonstramos que este modelo de consistência é uma boa escolha para diminuir o *overhead* imposto pelo protocolo HLRC de coerência de cJava.

Uma desvantagem da implementação do protocolo HLRC é de ocasionar uma distribuição de dados desbalanceada, devido a interferência negativa da política de atribuição de residência com a semântica da linguagem Java. Além disto, seria mais eficiente usar um sistema de memória compartilhada distribuída que adotasse granularidade de coerência mais fina.

As características da linguagem, como o fornecimento de classes *thread* e mecanismo para sinalização e sincronismo entre *threads*, simplificaram a implementação de cJava. Porém o mais importante é que nossa implementação não adicionou novas características à linguagem de programação Java.

Como trabalhos futuros, o conflito existente entre a implementação do protocolo HLRC e a semântica da linguagem Java deverá ser eliminado para melhorar o desem-

penho de cJava. Além disto, uma nova versão do ambiente cJava deverá utilizar a máquina virtual Java 1.4.

Os resultados experimentais nos permitem concluir que a estratégia de alterar a máquina virtual Java para suportar execução de aplicações *multithread* em um *cluster* é viável e promissora, porém mais experimentos serão necessários em trabalhos futuros para avaliar as sugestões propostas.

Referências Bibliográficas

[1] Adve, Sarita V., Gharachorloo, Kourosh. "Shared Memory Consistency Models: A Tutorial". *IEEE Computer*, v. 29, n. 12, pp. 66–76, Dec. 1996.

[2] Amza, C., Cox, A. L., Dwarkadas, S., et al. "TreadMarks: Shared Memory Computing on Networks of Workstations". *IEEE Computer*, v. 29, n. 2, pp. 18–28, Jan. 1996.

[3] Aridor, Yariv, Factor, Michael, Teperman, Avi. "cJVM: a Single System Image of a JVM on a Cluster". In: *Proceedings of the International Conference on Parallel Processing*, pp. 4–11, Wakamatsh, Japan, Sep. 1999.

[4] Arnold, Ken, Gosling, James, Holmes, David. *The Java Programming Language*. 2 ed., California, USA, Addison Wesley, 2000.

[5] Chen, X., Allan, V. H. "MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines". In: *Proceedings of the International Conference on Parallel and Distributed Processing Technique and Applications*, pp. 91–98, Las Vegas, Nevada, USA, Jul. 1998.

[6] "CORBA Technology and the Java Plataform Homepage". `http://java.sun.com/j2ee/corba/`; accessed December 2, 2002.

[7] Iftode, Liviu. *Home-Based Shared Virtual Memory*. PhD thesis, Princeton University, Princeton, USA, Jun. 1998.

[8] "Java Remote Method Invocation Homepage". `http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html`; accessed December 2, 2002.

[9] "Java Sockets Documentation Homepage". `http://java.sun.com/docs`; accessed December 2, 2002.

[10] Judge, Alan, Nixon, Paddy, et al. "Distributed Shared Memory". In: Buyya, Rajkumar (editor), *High Performance Cluster Computing*, v. 1, 1 ed., chapter 17. New Jersey, USA, Prentice Hall PTR, 1999.

[11] Keleher, P., Cox, Alan, Zwenepoel, Willy. "Lazy Release Consistency for Software Distributed Shared Memory". In: *Proceedings of the 9th International Symposium on Computer Architecture*, pp. 13–21, Gold Coast, Australia, May 1992.

[12] Keleher, Peter, Cox, Alan L., et al. "TreadMarks: Distribuited Shared Memory on Standard Workstations and Operating Systems.". In: *Proceedings of the Winter USENIX Conference*, pp. 115–132, San Francisco, California, USA, Jan. 1994.

[13] Leong, M., Cheung, O. "A bit-serial Implementation of the International Data Encryption Algorithm IDEA". In: *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*, pp. 121–131, Napa, California, USA, Apr. 2000.

[14] Li, K., Hadak, P. "Memory Coherence in Shared Virtual Memory System". *ACM Transactions on Computer Systems*, v. 7, n. 4, pp. 321–359, Nov. 1989.

[15] Liang, Sheng, Bracha, Gilad. "Dynamic Class Loadding in Java Virtual Machine". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 36–44, Vancouver, Canada, Oct. 1998.

[16] Lindholm, Tim, Yellin, Frank. *The Java Virtual Machine Specification Second Editon.* 2 ed., California, USA, Addison Wesley, 1999.

[17] Lobosco, Marcelo, Amorim, Claudio, Loques, Orlando. "Java for High-Performance Network-based Computing: a Survey". *Concurrency and Computation: Practice and Experience*, v. 14, n. 1, pp. 1–31, Jan. 2002.

[18] Microsystems, Sun. *Making Java Work for High-End Computing*. Technical Report JGF-TR-1, Sun Microsystems, 1998.

[19] "Performance Counter Library". `http://fz-juelich.de/zam/PCL`; accessed November 12, 2002.

[20] Protic, Jelica, Tomasevic, Milo, Milutinovic, Veljko. "Distributed Shared Memory: Concepts and Systems". *IEEE Parallel e Distributed Technology*, v. 4, n. 2, pp. 63–79, Jun. 1997.

[21] Rangarajan, Muralidharan, Iftode, Liviu. "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance". In: *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 341–352, Atlanta, Georgia, USA, Oct. 2000.

[22] Samanta, Rudrajit, Bilas, Angelos, et al. "Home-based SVM Protocols for SMP Clusters: Design and Performance". In: *Proceedings of the 4th Symposium on High-Performance Computer Architecture*, pp. 1–13, Las Vegas, USA, Fev. 1998.

[23] Shi, Weisong, Hu, Weiwu, Tang, Zhimin. *Shared Virtual Memory: A Survey*. Technical Report 980005, Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences, 1998.

[24] Stenstrom, Per. "A Survey of Cache Coherence Schemes for Multiprocessors". *IEEE Computer*, v. 23, n. 6, pp. 12–24, Jun. 1990.

[25] Stevens, W. Richard. *Unix Network Programming*. 1 ed., New Jersey, USA, Prentice Hall, 1990.

[26] Veldema, R., Bhoedjang, R., H., Bal. *Distributed Shared Memory Management for Java.* Technical report, Faculty of Sciencies, Vrije Universiteit, Amsterdam, 1999.

[27] Venners, Bill. *Inside the Java 2 Virtual Machine.* 2 ed., New York, USA, Mc Graw Hill, 1999.

[28] "Virtual Interface Architecture Specification". `http://www.viarch.org`; accessed November 12, 2002.

[29] Woo, Steven Cameron, Ohara, Moriyoshi, Torrie, Evan, Singh, Jaswinder Pal, Gupta, Anoop. "The SPLASH-2 Programs: Characterization and Methodological Considerations". In: *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36, Santa Margherita Lugure, Italy, Jun. 1995.

[30] Yu, Weimin, Cox, Alan. "Java/DSM: A Platform for Heterogeneous Computing". *Concurrency: Practice and Experience*, v. 9, n. 11, pp. 1213–1224, Nov. 1997.